Multi-Version-PulsatingSTM: A Multi-Version Optimistic Concurrency Control Scheme for Highly Parallel in-Memory Workload in a Multi Core Environment

Sana Jafar, Ranjana Rajnish, Pankaj Kumar

Abstract: ***Large in-memory data structures have a significant application in the fields of graphics, gaming, military and all the possible areas where Big Data can be employed. Their fame in the area of science and technology is attributable to fast in-memory access by the processor as compared to on-disk data structures. These enormous data structures can be accessed still fast and efficiently through parallel computing. For employing highly parallel computations, equally parallel algorithms are required. One of the most desirable attributes of such algorithms is their ability to control concurrency and avoid any deadlocks while being time and energy efficient. This paper presents a multi-version optimistic concurrency control algorithm based on timestamping. This algorithm is lock free and is tested on 64 simulated CPU cores on a multi core simulator. The algorithm is a Software Transactional Memory approach employing 16, 32, 40 and 50 threads in different tests running on the simulator. Half of the threads are doing reading and half are doing writing operation in each case while accessing an in-memory dynamic array. Being lock free and employing lazy timestamp calculations, this approach is better than other existing concurrency control approaches.***

Keywords: Software Transactional Memory, Optimistic Concurrency Control, in-memory data structures, Timestamping, multi-version, sniper multi-core simulator, multi cores, Cycles Per Instructions

I. INTRODUCTION

Parallelism of transactions or threads is important for faster access of enormously large files and data structures. But with such large parallelism, there is a necessity of effective parallelism and concurrency control. A lot of work is being done in this direction. Some of it covers locking mechanisms, barriers, time stamping, optimistic and multi version concurrency control and Transactional Memory approach.

[1] explains the basic concurrency control techniques like locks, timestamping, optimistic and multi version concurrency control. There are several famous in-memory multi version concurrency control schemes like Hekaton[2], Hyper[3], Bohm[4], Deuteronomy[5] and ERMIA[6]. MOCC[7], Cicada[8] are some of the recent multi-version optimistic concurrency techniques. TicToc[9], is a multi version optimistic and timestamp ordering scheme. Whatever may be the technique, locks are used in some or the others phases in order to achieve concurrency control. [10-15] shows software transactional memory approach wherein locks are employed in the validation phase.

It is seen that locking is the inherent technique of all the concurrency control mechanisms. But locking has its disadvantages. The most obvious one is that, it limits the concurrency by allowing just one thread to enter the critical section at a time leading to the possibility of a deadlock.

Not just the use of locks, but scalability of these techniques with increasing number of threads or transactions is also a matter of concern. In [15] authors have proposed a novel software transactional memory approach for NUMA architectures. Here, the authors have obtained throughput by running various TM algorithms, namely, TL2[16], SwissTM[17], TinySTM[18], RingSTM [19] and NOrec[20] on a 64 cores AMD commercially available server but have observed no improvement in throughput as the number of threads increase beyond 15.

In this paper, the authors have developed a multi version optimistic concurrency control technique based on Software Transactional Memory approach and timestamping. This technique is an extension of the optimistic concurrency control technique based on Software Transactional Memory and timestamping for in-memory data structures in multi core systems developed by the authors of this paper[21]. The technique possess following attributes:

1. Being lock free, it provides better concurrency among executing threads and avoids deadlocks.

2. As it is a Software Transactional Memory approach, it follows all the three attributes of transactions in in-memory systems namely Atomicity, Consistency and Isolation.

3. The timestamping mechanism used here is not centralized. Each transaction as it enters the system, gets its own timestamp by calling a procedure. Each valid writing transaction calculates its commit timestamp by the read timestamp of the element in its write set. This eliminates the possibility of any bottlenecks arriving due to centralized

Revised Manuscript Received on October 15, 2019
***** Correspondence Author
Sana Jafar*, Amity Institute of Information Technology, Amity University Uttar Pradesh, Lucknow Campus, Lucknow(India), India. Email: india.sana@gmail.com
Ranjana Rajnish, Amity Institute of Information Technology, Amity University Uttar Pradesh, Lucknow Campus, Lucknow(India), India. Email: rrajnish@lko.amity.edu
Pankaj Kumar, Department of Computer Science and Engineering, Sri Ram Swaroop College of Engineering and Management, Lucknow, India. Email: pk79jan@gmail.com

timestamp manager.

4. Also as the commit timestamp of each transaction is calculated only at the time of final write operation this allows for maximum transactions to perform write operation in their private sets.

5. The multi version attribute of this algorithm allows writing the multiple versions of the same element in a dynamically growing data structure by the valid transactions and allowing their access by other transactions.

II. MULTIVERSION-PULSATING-STM

Multi-Version-PulsatingSTM is an extension of the PulsatingSTM algorithm[21] already developed by the authors of this paper. It is an optimistic concurrency control algorithm based on Software Transactional Memory approach and timestamping. Multi-versioning and having distinct reading and writing transactions are the extensions employed in the algorithm where each valid writing transaction is allowed to write a separate version of the same element on the original data structure and a valid reading transaction is used to read and change the read timestamp of the element in the original data structure.

Multi-Version-PulsatingSTM has the same three phases namely Read Phase, Validation Phase and Write Phase as the PulsatingSTM:

A. Read Phase

The Read Phase of the algorithm has the following steps:

- Each transaction with an even ID, is a writing transaction. It first copies an element from the data structure in its write set and does the update there. Then it copies the same element in its read set. Transactions with an odd ID are reading transactions and they simply copy the elements in their read set.
- Both type of transactions note down the read and write timestamps of the element in their read/write sets.
- They set the pointer to the location of the element in the original data structure.

B. Validation Phase

After all the transactions have read the elements in their read/write sets, depending upon the operation, the validation phase arrives. In the validation phase two major things happen:

- It is decided using the following steps whether or not the transaction is a valid transaction.
- For the valid writing transactions the commit timestamp is calculated.
- For the valid reading transactions, the read timestamp of the elements read is modified to the transaction's timestamp.

Validation Phase has the following steps:

1. If any one of the following conditions holds, then the transaction is not a valid transaction and it has to rollback and abort. For such transactions commit timestamp is not generated.
   - The read timestamp and the write timestamp of the element in the transaction's read set are equal.
   - The read timestamp and the write timestamp of the element in the transaction's read set are having a difference of 1 unit.
   - The write timestamp is greater than the read timestamp in the read set.
2. If none of the above conditions hold then the transaction is a valid transaction and commit timestamp for the valid writing transaction is computed as below in point 3. However, if it is a valid reading transaction then the read timestamp of the element read in its read set is set to the timestamp of this transaction.
3. The valid writing transaction's timestamp is compared with the read and write timestamp of the element in its read set and checked whether or not it is in between the write and read timestamp of the element.
4. If the transaction's timestamp is not in between the read and write timestamp of the element in the read set then it is altered to satisfy the constraint.
5. Now the commit timestamp is finally computed by making the altered timestamp from point 4 greater than the read timestamp of the element in the write set of the transaction.

C. Write Phase

Once the transaction is decided to be a valid writing transaction with a commit timestamp, then its write set is written on the original data structure. If there are multiple valid transactions then each transaction is allowed to write a separate version on the original data structure.

In case the valid transaction is a reading transaction, then its read set is copied to the original data structure.

III. DESIGN ELEMENTS OF MULTI-VERSION-PULSATINGSTM

Multi-Version-PulsatingSTM has the following design elements:

1. A dynamic one dimensional array as global data structure to be accessed by all the transactions.
2. Each element in the array has some metadata that is comprised of read and write timestamp, a data value and a pointer to the element in the original data structure. The read timestamp is the timestamp of the valid reading transaction that has recently read that element and write timestamp is the commit timestamp of the valid writing transaction that has currently updated that element. The metadata is tabularized in [21].
3. Each writing transaction has a private read and write set. Each reading transaction just has a private read set.
4. Whenever a transaction has to perform the read operation on some element, it copies that element in its read set, notes down its read and write timestamps and the data value, makes the pointer point to the original element in the data structure.
5. Whenever a transaction has to perform the write operation on some element, it copies that element in its write set, does the update or writing operation on the data value, notes down the read and write timestamp of the element and, makes the pointer point to the original element in the data structure.

Algorithm 1 demonstrates the Transaction Begin, Read Phase, Validation Phase and Write Phase of this algorithm.

Algorithm 1 Multi-Version-PulsatingSTM algorithm

```
1: procedure BeginTX
2:  timestamp ⟵ autoinc()
3:  ID ⟵ omp_get_thread_num()
4: end procedure
5: procedure ReadTX
6:  if ID % 2==0 do
7:  p ⟵ write(tranwrite, (arr+0),0,p,timestamp)
8:  s ⟵ read(tranread,(arr+0),0,s,timestamp)
9:  else
10:  s ⟵ read(tranread,(arr+0),0,s,timestamp)
11: end procedure
12: s ⟵ s – 1;  p ⟵ p -1;
13: procedure ValidateTX
14:  k ⟵ 0
15:  if tranread[k].rtime != tranread[k].wtime AND
     tranread[k].rtime –tranread[k].wtime != 1 AND
     tranread[k].wtime < tranread[k].rtime  do
16:  while k<s do
17: while timestamp<=tranread[k].wtimeOR
     timestamp>=tranread[k].rtime do
18: if timestamp<=tranread[k].wtime do
19:  timestamp++;
20:  elseif  timestamp>=tranread[k].rtime do
21:timestamp--;
22:  end if
23:  end while
24:  k ⟵ k+1
25:  end while
26:  end if
27:  k ⟵ 0
28: while k<s do
29:  if  timestamp >tranread[k].wtime  AND
     timestamp<tranread[k].rtime  do
30:  flag ⟵ 1;
31:  else
32:  flag ⟵ 0; break;
33:  end if
34:  k ⟵ k+1
35: end while
36: if flag == 0 do
37:  Transaction has read invalid version and has to
     roll back
38: for  j ⟵ 0, j< p do
39  if tranread[k].point==tranwrite[j].point do
40:  for l ⟵ j,  l<p-1 do
41:  tranwrite[j] ⟵ tranwrite[j++];
42:  l ⟵ l + 1
43:  end for
44:  p ⟵ p -1; break
45:  end if
46: j ⟵ j+1
47: end for
48: else do
49: Transaction has read a valid version
50:  k ⟵ 0
51: if  p > 0 do Transaction is a writing transaction
52:  while k < p do
53:  if  timestamp < tranwrite[k].rtime do
54:  timestamp ⟵ tranwrite[k].rtime
55:  end if
56:  k ⟵ k+1
57:  end while
58:  if  k == p do
59:  timestamp ⟵ timestamp +1
60:  end if
61:  commit timestamp is timestamp
62: else Transaction is a reading transaction
63:  for k=0 , k < s do
64:  tranread[k].rtime ⟵ timestamp;
65:  *(tranread[k].point) ⟵ tranread[k];
66:  k ⟵ k+1
67:  end for
68: end if
69: end if
70: end procedure
71: procedure WriteTX
72:  for  j=0 , j<p do
73:  tranwrite[j].wtime ⟵ timestamp
74:  tranwrite[j].rtime ⟵ timestamp
75:  count ⟵ count+1; ind ⟵ count;
76:  Dynamically incrementing the size of array arr by
     ind
77:  *(arr+(ind-1)) ⟵ tranwrite[j];
78:  j ⟵ j+1
79: end for
80: end procedure
```

Here, **count** and **arr** are global variables. **count** is initialized to 1 and **arr** is the dynamic integer array. **arr** is the array in which the transactions are trying to access the elements concurrently for reading and writing. **count** is used to maintain the count of the versions written to **arr**. **tranwrite**, **tranread**, **s**, **p**, **timestamp, ind, ID** are the private variables of each transaction. **tranwrite** and **tranread** are the private write set and read set respectively of each transaction. **s** and **p** are the size of **tranread** and **tranwrite** respectively. **timestamp** is the private variable for holding unique timestamp of each transaction. **ind** is the private variable that is used to increment **arr** by for every new version created by a transaction. **ID** is the unique number allotted to every thread in the system.

Algorithms 2, 3 and 4 demonstrate the read( ) write( ) and autoinc( ) functions respectively used in Algorithm 1.

Algorithm 2 Read operation algorithm

```
1: procedure read (struct element *TR, struct element
*v,int i,int size, int timestamp)
2: TR[i]←*v;
3: size ←size+1;
4: TR[i].rtime ← v->rtime;
5: TR[i].wtime ← v->wtime;
6: TR[i].point ← v;
7: return size;
8: end procedure
```

Algorithm 3 Write operation algorithm

```
1: procedure write (struct element *TW, struct
element *v,int i,int size, int timestamp)
2: size ← size+1;
3: TW[i] ← *v;
4: update (TW[i].data);
5: TW[i].rtime ← v->rtime;
6: TW[i].wtime ← v->wtime;
7: TW[i].point ←v;
8: return size;
9: end procedure
```

Algorithm 4 autoinc algorithm

```
1: procedure autoinc
2: static int c ←1;
3: c ←c +1;
4: return c;
5: end procedure
```

In the algorithm 1, line number 1 to 3 is BeginTX procedure indicating the Beginning phase of the algorithm. During this time each parallel transaction is allotted a unique ID and a timestamp.

Line numbers 5 to 11 is ReadTX procedure which indicates the reading phase of the algorithm. During this phase, the transactions with even ID are supposed to perform both writing and reading operations by calling read() and write () functions respectively. However, transactions with odd ID are supposed to perform just reading operation by calling read() function only.

Line number 13 to 70 is ValidateTX procedure. The validation takes place in the way explained in sub-section B of section II above.

In number 51, while checking the value of p to be greater than 0, it is decided whether or not the transaction has a non-empty write set. If the write set of the transaction is non-empty, it is decided that it is a writing transaction which has written at least one element to its write set. However, if the write set is empty then it can be safely decided that the transaction is a reading transaction and therefore there is no element in its write set.

From line number 52 to 61, the commit timestamp of the writing transaction is calculated. From line number 63 to 66, reading transaction updates the read timestamp of the element in its read set to its own timestamp, and copies its readset to the original data structure.

Line number 71 to 80 is the WriteTX procedure. The valid writing transaction first updates the read and write timestamps of the elements in its writeset and then writes the updated data value as a new version in the original data structure. For this it first dynamically increases the size of the data structure.

IV. EXPERIMENTATION AND RESULT ANALYSIS

The above algorithms are implemented in OpenMP using C. They are tested on sniper-6.1[22] using the gainestown configuration.

The gainestown configuration has the following settings:

- Core frequency—2.66 GHz
- Number of cores sharing L3 cache— 4
- Data access time by L3 cache – 30 cycles
- Network memory model --- bus
- Bus bandwidth – 25.6 GB/s (12.8 GB/s per direction and per connected chip pair)

Local traffic has been ignored because the memory controllers are on chip.

Authors have executed and tested the OpenMP C code for Multi-Version-PulsatingSTM on 64 simulated cores employing 16, 32, 40 and 50 threads consecutively.

The average Cycles Per Instruction (CPI) graphs obtained are shown below in Fig. 1-4.

These graphs are plotted with time in microseconds on X-axis versus CPI percentage on y-axis.

The spikes in the graphs are representing read phase and the heap towards the right side is representing validation and the write phases.

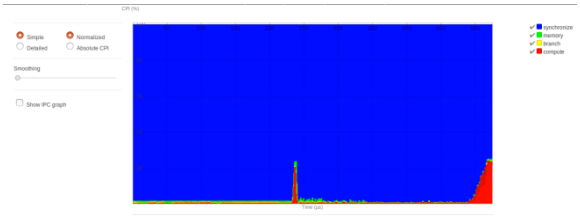

Fig. 1. Average CPI graph for the algorithm running 16 threads on 64 simulated cores

Fig. 1 shows the result of running the proposed algorithm on 64 simulated CPU cores employing 16 threads. The read phase is spiking up at 232 microseconds. The validation and the write phase starts from 466 microseconds till 527 microseconds and cover around 61 microseconds of the graph. In terms of CPI percentage, read phase occupy 25% of CPI and validation and write phase also occupy maximum 25% CPI. The total time spend in running this algorithm is 526.4 microseconds.

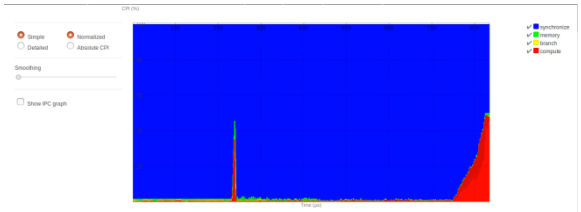

Fig. 2. Average CPI graph for the algorithm running 32 threads on 64 simulated cores

Fig. 2 shows the result of running the proposed algorithm on 64 simulated CPU cores employing 32 threads. The read phase is spiking up at 232 microseconds and covers around 45% CPI. The validation and write phase also cover maximum 45% CPI. They start from 750 microseconds till 836 microseconds and cover around 86 microseconds of the graph. The total time occupied in running this algorithm is 836 microseconds.

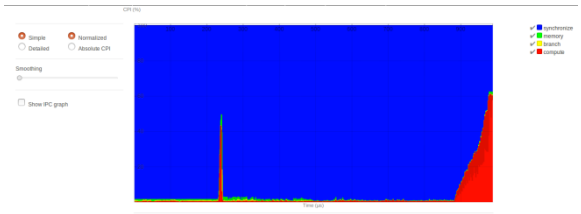

Fig. 3. Average CPI graph for the algorithm running 40 threads on 64 simulated cores

Fig. 3 shows the result of running the proposed algorithm on 64 simulated CPU cores employing 40 threads. The read phase is spiking up at 232 microseconds and occupies 50% CPI. The validation and write phase starts at 875 microseconds and stretch upto 990 microseconds, covering 115 microseconds. Also the maximum CPI percentage of these two phases is 62%. The total time taken in running this algorithm is 989.7 microseconds.

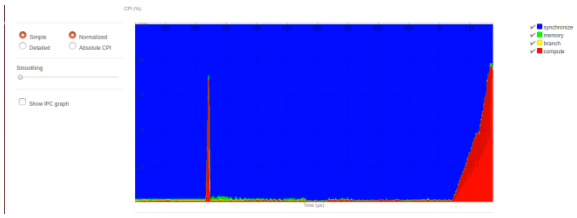

Fig. 4. Average CPI graph for the algorithm running 50 threads on 64 simulated cores

Fig. 4 shows the result of running the proposed algorithm on 64 simulated cores employing 50 threads. The total time taken by this algorithm is 1.178 milliseconds. The read phase spikes up at 240 microseconds and consumes 72% CPI. The validation and write phase starts at 1.048 milliseconds and stretches up to 1.178 milliseconds. These two phases consume maximum of 79% CPI.

Metrics observed after running the proposed algorithm on increasing number of threads are tabulated below:

Table-I: Parametric values from sniper for running Multi-Version-PulsatingSTM employing different number of threads on 64 cores

	Threads			
	16	**32**	**40**	**50**
Instructions	3.704 m	12.94 m	20.06 m	29.84 m
IPC	0.042	0.091	0.119	0.149
Cycles	1.408 m	2.224 m	2.633 m	3.133 m
Time	526.4 µs	836 µs	989.7 µs	1.178 ms
Branch MPKI	2.058	1.337	0.993	0.781
L1-I MPKI	1.502	0.799	0.631	0.519
L1- D MPKI	1.920	0.915	0.692	0.553
L2 MPKI	3.159	1.625	1.264	1.032
DRAM APKI	1.366	0.614	0.468	0.378

IPC: Instructions Per Cycle, MPKI: Misses Per Kilo Instructions, L1-I: Instruction level L1 Cache, L1-D: Data level L1 Cache, L2: L2 cache, DRAM: Dynamic Random Access Memory, APKI: Access Per Kilo Instructions

From the Table 1, it is clear that as the number of threads is increasing, the cache misses are reducing as well as the DRAM access per kilo instructions is also reducing thus giving enhanced throughput. Similar observation is made by running PulsatingSTM on 64 simulated CPU cores employing 16, 32, 40 and 50 threads.

Fig. 5 shows the graph of throughput versus number of threads. It clearly shows that as the number of threads increase, throughput also increases.

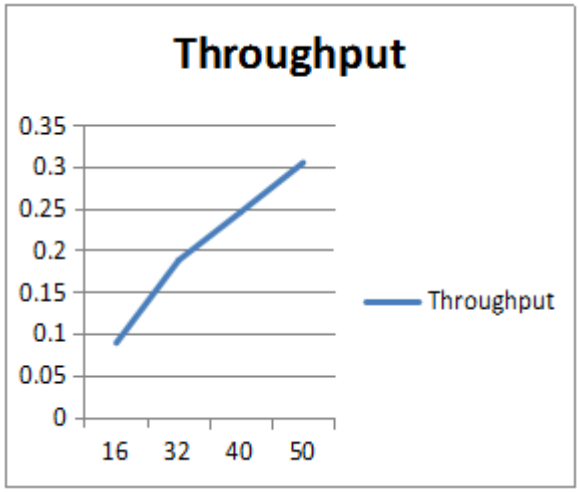

Fig. 5. Throughput Versus Number of Threads

V. CONCLUSION AND FUTURE SCOPE

Multi-Version-PulsatingSTM is an extension of PulsatingSTM algorithm[21] that is developed by the authors of this work. The extension is based upon employing half of the transactions doing reading operation while half doing writing operation concurrently on a shared in-memory data structure, and the valid writing transactions are writing the updated data element as a different version on the shared data structure. The concurrency among the transactions is controlled by a novel Software Transactional Memory based optimistic concurrency control technique employing timestamping.

The throughput of the algorithm is found to be improving with the increasing number of threads. As it is a lock-free approach it is undoubtedly better than many lock based STM algorithms in literature.

The authors propose to employ this algorithm as a means to perform parallel sorting in an enormously large size data structure. Also, the authors propose to modify this algorithm to work on other data structures like tree and linked list.

REFERENCES

1. Sana Jafar, Pankaj Kumar, Ranjana Rajnish, "Reviewing the Current Concurrency Control Techniques in Multi and Many core systems", In Proceedings of the 12th INDIACom; INDIACom-2018 5th 2018 International Conference on "Computing for Sustainable Global Development", Bharati Vidyapeeth's Institute of Computer Applications and Management (BVICAM), New Delhi (INDIA), March 14th – 16th, 2018, pp. 525-530.
2. C. Diaconu, C. Freedman, E. Ismert, P.-A. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling. Hekaton, "SQL Server's memory-optimized OLTP engine", In Proceedings of the 2013 SIGMOD International Conference on Management of Data, New York, USA, June 22, 2013, pp. 1243-1254.
3. T. Neumann, T. Mühlbauer, and A. Kemper. "Fast serializable multi-version concurrency control for main-memory database systems", In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31-June 04, 2015, pp. 677-689.
4. J. M. Faleiro and D. J. Abadi., "Rethinking serializable multiversion concurrency control", In. Proceedings of VLDB Endowment, Vol.8, No. 11, July , 2015. pp. 1190-1201.
5. J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang, ".High performance transactions in Deuteronomy". In Proceedings of Conference on Innovative Data Systems Research (CIDR 2015), Asilomar, California, USA, Jan. 4-7, 2015.
6. K. Kim, T. Wang, R. Johnson, and I. Pandis. "ERMIA: Fast memory-optimized database system for heterogeneous workloads". In Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data, San Francisco, California, USA, June 26- July 01, 2016, pp. 1675-1687.
7. T. Wang, and H. Kimura, "Mostly-Optimistic Concurrency Control for Highly contended dynamic workloads on a thousand cores", In proceedings of VLDB Endowment, vol. 10. No. 2., 2016, pp. 49-60.
8. H. Lim, M. Kaminsky, and D.G. Andersen, "Cicada: Dependably Fast Multi-core In-Memory Transactions", In Proceedings of the 2017 ACM International Conference on Management of Data SIGMOD, Chicago, Illinois, USA, May 14 - 19, 2017, pp. 21 – 35.
9. X. Yu, A. Pavlo, D. Sanchez, and S. Devadas, "TicToc: Time travelling Optimistic Concurrency Control", In Proceedings of the 2016 International Conference on Management of Data SIGMOD, San Francisco, California, USA, June 26 - July 01, 2016, pp. 1629-1642.
10. Nir Shavit and Dan Touitou, "Software Transactional memory." In Proceedings of the 14th Annual ACM Symposium of PODC 95, Ottawa Ontario CA, August 20-23, 1995, pp. 204-213.
11. El-Shambakey, Mohammed and Binoy Ravindran, "STM concurrency control for multicore embedded real-time software: time bounds and tradeoffs." In Proceedings of SAC (2012), Riva del Garda, Italy, March 25-29, 2012, pp. 1602-1609.
12. Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, Benjamin Hertzberg, "McRT-STM: A High Performance Software Transactional Memory System for a Multi-Core Runtime.", In Proceedings of 11th ACM SIGPLAN symposium on PPoPP, New York, NY, USA., '06 March 29-31, 2006, pp. 187-197.
13. Yunlong Xu, Rui Wangy, Nilanjan Goswamiz, Tao Liz, Lan Gaoy, Depei Qian, "Software Transactional Memory for GPU Architectures", In Proceedings of IEEE/ACM International Symposium on CGO '14, Orlando, FL, USA, February 15 - 19 2014, pp. 1
14. Xiaowei Ren and Mieszko Lis, "High-performance GPU Transactional Memory via Eager Conflict Detection", In Proceedings of 2018 International Symposium on High Performance Computer Architecture, Vienna, Austria, Feb 24-28, 2018, pp. 235-246
15. Mohamed Mohamedin, Sebastiano Peluso, Masoomeh Javidi Kishi, Ahmed Hassan, Roberto Palmieri " Nemo: NUMA-aware Concurrency Control for Scalable Transactional Memory", In Proceedings of 47th International Conference on Parallel Processing, Eugene, OR, USA, August 13–16, 2018, Article No. 38.
16. Dave Dice, Ori Shalev, and Nir Shavit, "Transactional Locking II.", In Proceedings of the 20th international conference on Distributed Computing, Stockholm, Sweden, September 18 - 20, 2006 , pp. 194–208.
17. Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka, " Stretching Transactional Memory", In Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, Ireland, June 15 - 21, 2009, pp. 155-165.
18. Pascal Felber, Christof Fetzer, and Torvald Riegel, "Dynamic Performance Tuning of Word-based Software Transactional Memory", In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, Salt Lake City, UT, USA, February 20 - 23, 2008, pp. 237–246.
19. Michael F. Spear, Maged M. Michael, and Christoph von Praun, "RingSTM: Scalable Transactions with a Single Atomic Instruction", In Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures, Munich, Germany, June 14 - 16, 2008, pp. 275–284.
20. Luke Dalessandro, Michael F. Spear, and Michael L. Scott, "NOrec: Streamlining STM by Abolishing Ownership Records". In Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Bangalore, India, January 09 - 14, 2010, pp. 67–78.
21. Sana Jafar, Ranjana Rajnish, and Pankaj Kumar, "PulsatingSTM-The in-memory Optimistic Concurrency Control Technique for Multi core systems (Journal style—Submitted for publication)," *International Journal of engineering and Advanced Technology (IJEAT)*, Vol-9(Issue-1) to be published.
22. T. E. Carlson, W. Heirman, and L. Eeckhout., "Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations", In Proceedings of International Conference on High Performance Analysis, Networking, Storage and Analysis, Seatle, WA, USA, Nov. 12-18, 2011, pp. 1-12.

AUTHORS PROFILE

Sana Jafar is currently working as an IT consultant with Argus Technology LLC. She is a research scholar in the faculty of Information Technology from Amity University Uttar Pradesh Lucknow Campus, enrolled since January 2015. She has worked as an Assistant Professor (Computer Science & IT) in the Department of Amity School of Engineering and Technology, Amity University Uttar Pradesh Lucknow Campus from 2009 till 2018. She completed her MCA with silver medal and received her degree with honors in 2009. Her area of research is Parallel Computing and High Performance Computing. She is a student member of IEEE. She has 4 papers published and presented in IEEE sponsored International and National conferences and one book chapter published in Scopus Indexed Ebook series titled "Advances in Parallel Computing", IOS Press, Netherlands. Sana Jafar has Participated in the Short Term Course (under QIP IIT Delhi) on many core parallel Programming at IIT Delhi from 4th June -15th June 2018., learning hands on Nvidia CUDA: API for parallel programming in GPU based architecture and accessed the HPC clusters at IIT Delhi (PADUM). She has also worked as an intern under Prof Subodh Kumar (Dept. CSE at IIT Delhi) under the Summer Faculty Research Fellow Program from 4th June -13th July 2018 at IIT Delhi. She has published a useful workbook on Object oriented programming using C++ as main author(publishers: Alok Prakashan) for the B.Tech students of Amity University and is in the process of generalizing it for the B.Tech pursuing students of all the engineering colleges in Uttar Pradesh. She has successfully attended various faculty development programs and workshops in Amity University Lucknow campus and outside. As well has played an important part in conducting such programs within the Amity University Lucknow campus. She has attended the five days military training camp organized by Amity University Manesar in 2016 as faculty guide with post graduate students. She has also secured a second position in women badminton in the annual sports meet of Amity University Lucknow (Sangathan) in 2015.

Sana Jafar is diligently working towards inventing innovative and efficient ways for improving concurrency control methods in multi and many core systems using STM and optimistic methods.

Dr. Ranjana Rajnish is an Assistant Professor at Amity Institute of Information Technology at Amity University, Lucknow. Dr. Ranjana possesses approximately 25 years of experience in academics/research. She has been engaged with institutions like U.P. Technical University and Amity University in roles ranging from a faculty in computer science to Academic Head. Her area of interest includes Software Engineering, Opinion

Mining/Sentiment Analysis and Healthcare.
She has several publications in national and international journals and conference proceedings of National and International Conferences of repute. She is also member of various professional bodies like Computer Society of India (CSI), Association of Computing Machinery(ACM), International Association of Engineers (IAENG), Internet Society and Computer Science Teaching Association (CSTA).
Along with being a committed teacher and a passionate researcher, Dr. Ranjana is reviewer for various International Journal and member of editorial board for different International Journals. She is also reviewer, member of technical programme committee in various conferences of repute in and outside India. She has many Ph.D. scholars pursuing Ph.D. under her.

Dr. Pankaj Kumar is currently working as Assistant Professor (Reader) in Department of Computer Science & Engineering in Sri Ramswaroop Group of Professional College, Lucknow. He has more than 18 years of teaching experiences. He received his MCA degree in 2001, M.Tech in 2010 and PhD degree in Computer Application in 2011. His Area of Expertise is Parallel Computing/ Mining/Security. More than 50 research papers of Dr. Pankaj Kumar have been published in various national/international journals and IEEE proceeding publication. He is Senior Member of IEEE, Professional Member of ACM and Life member of CSI, IETE, ISTE, IAENG, ISOC and IACSIT. He is member of Management Committee of CSI and IETE Lucknow Chapter. He is reviewer for various International Journal and member of editorial board for different International Journals. He also participated in various conferences as reviewer, member technical committee, and co-chair. One PhD thesis is awarded and eight students are enrolled as PhD scholar under his guidance. More than 10 students are guided by him in M.Tech Thesis.